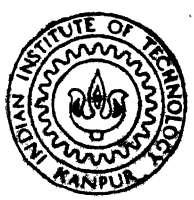
AN EXPERIMENTAL MULTIDATABASE SYSTEM

by

K. N. KUMAR



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

MARCH, 1993

AN EXPERIMENTAL MULTIDATABASE SYSTEM

A Thesis Submitted
In Partial Fulfilment of the Requirements
for the Degree of
MASTER OF TECHNOLOGY

by

K N KUMAR

to the

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERIN
INDIAN INSTITUTE OF TECHNOLOGY
KANPUR

March, 1993

CERTIFICATE

This is to certify that the work contained in this thesis titled **AN EXPERIMENTAL MULTIDATABASE SYSTEM** by **K N KUMAR**, has been carried out under my supervision and has not been submitted elsewhere for a degree.

T. V. PRABHAKAR
Assistant Professor

Date: 2 Mar 93

CERTIFICATE

This is to certify that the work contained in this thesis titled **AN EXPERIMENTAL MULTIDATABASE SYSTEM** by **K N KUMAR**, has been carried out under my supervision and has not been submitted elsewhere for a degree.

T V Prabhakar

T. V. PRABHAKAR
Assistant Professor

Date: 2 Mar 93

Acknowledgements

I express my deep gratitude to my thesis supervisor Dr.T.V.Prabhakar for his excellent guidance throughout the course of my thesis work. I thank him for his cooperation and encouragement when it was needed most.

I also thank Dr.S.Sadagopan for his kind assistance.

I take this opportunity to thank all my friends, from Anandar to Vishwanath, for their convivial companionship. I wish to express special thanks to Madhu, Jayateerth, and (Oh) Galagali, all of whom have taught me much during my stay here. I acknowledge happy memories of 'Ambi' and 'CR'.

I also thank the staff of CC for their cooperation.

K N Kumar
Kanpur
March, 1993.

Contents

Abstract

Recent advances in information technology have increased the need for users to share information. This means that users must be provided access to multiple databases even if they are under different database management systems and on different computers. Multidatabase systems act as a front-end to multiple databases. The fact that the databases can be heterogeneous in nature has given rise to languages for multidatabase systems. Such a language typically provides users facilities to access data from more than one database. In this thesis, we discuss the design and implementation of a feature for retrieving data from multiple databases with a single query. In the process, we also look at strategies for evaluating such an inter-database query.

Chapter 1

Introduction

The traditional approach to data processing has been to centralize the data of an enterprise under a single database management system (DBMS). The database approach distinguishes between data description and data manipulation to facilitate data classification. Such a separation enables a system to achieve a high degree of data independence thereby allowing easy extensibility to the system. There are other objectives of a DBMS which include providing procedural and non-procedural interfaces, minimal redundancy and storage space, data integrity and control of concurrent accesses. Database systems offer to the user, high-level primitives for data management and automatically perform some basic functions. They not only allow a non-redundant, unified representation of all data managed in an enterprise but also provide efficient methods to access and manipulate the data. Such database systems often serve critical functions and represent significant investment for organizations. In today's information age there is a great need for sharing information. This implies that a user must be provided access to several databases even if they are under different database systems or on different computers. In many cases, there is a need for information exchange on a regionwide basis for the convenience of users. This means that globally important information exists in separate local databases that may

be incompatible, thus making the existing data inaccessible to the general user. The incompatibility may arise because of different data models or access languages. Major advances in computer network technology has made it possible to connect the databases residing on multiple computers and to access the data across the network. This has led to the conjecture that the next level of computerization is global, distributed systems that can share information from all participating data stores (sites). It is unreasonable to expect all systems to be converted to a single common data model with a single access method. It is also unreasonable to expect users to remember multiple access paradigms in order to use the separate databases. Multidatabases provide users with a common interface to multiple databases with minimal impact on the existing function of these databases.

There is a growing need for user-friendly global information sharing in recent times. There are several solutions to this. User requirements, existing hardware and software, and the amount of resources available will decide which solution is appropriate for a given environment. The solutions can be classified according to how tightly the global system integrates the local DBMSs. A tightly coupled system means that the global functions have access to internal DBMS functions. Hence global functions may have priority over local functions so that local DBMSs do not have full control over local resources. In a more loosely coupled system, the global functions access the local functions through the DBMS user interface. Global synchronization and efficiency are consequently reduced. The most loosely coupled system has an application layer residing above the local DBMS user interface. Here the site autonomy is maximum for each local DBMS. The most tightly coupled information-sharing system is a distributed database system. A distributed database is a collection of logically related data, distributed across several machines interconnected by a computer net-

work. Distributed databases are typically designed in a top-down fashion. The local DBMSs are homogeneous, i.e. they use the same data model and present the same functionality. The system maintains a global schema which represents the total information in the system. Users can submit queries over the global schema. Since all the local DBMSs are alike in functionality, performance can be increased by exploiting parallel processing and redundancy capabilities of multiple machines.

Global-schema multidatabases are more loosely coupled than distributed databases in the sense that there is a user interface of the local DBMS. There is a close cooperation between sites in maintaining the global schema. These are typically designed in a bottom-up fashion and can integrate pre-existing databases without modifying them. Since the local schemas can be different, the system must provide mappings between different local schemas and the single global schema. The mappings may be simple as in the case of name mappings, or complex as in the case of structural mappings.

Multidatabase language systems are more loosely coupled in that there is no explicit description of the total information in the system. The multidatabase system supports all functions, e.g. query processing, by providing query language tools to integrate data from the separate databases. These systems can integrate preexisting databases which may be heterogeneous in nature. The heterogeneity may mean different data models or differing implementations of the same model. The language of the multidatabase system provides special functions to map data from the different databases to a model and representation meaningful to the user. Some semantic heterogeneity usually exists at the common model level in the sense that similar syntactic definitions may have different semantics. This is the consequence of the database administrator's autonomy to tailor data to his needs. The language should provide tools to

eliminate this heterogeneity. Finally, interoperable systems are the most loosely coupled systems for information exchange. Global function is restricted to data exchange through message passing and local systems may include other types of information stores like expert systems, etc.

One of the major issues which must be handled in a multidatabase system is that of query processing. When querying a relational database, the user typically specifies the desired result without specifying the access paths. The DBMS determines how to access the data by applying a query processing algorithm that produces an access plan for a given query. The objective of query processing is to execute queries in such a way so as to minimize the cost of execution. For a centralized database system, the cost is usually the sum of the CPU and IO costs. The problem of selection of the best access plan is known to be computationally intractable for general queries [Hevner87]. Hence heuristics are used to simplify the problem and obtain a quasi-optimal solution [Hurson91]. Thus the role of the query processor is to map a relational calculus query into a near-optimal sequence of lower-level operations acting on relations. Parameters which influence this phase of processing include the choice of physical storage model and the complexity of database operations. The optimization of queries can be static or dynamic. Static optimization is done before the execution of a query. The main advantage is that it is performed once for many executions of the same query. This is important when database queries are embedded in application programs. On the other hand, dynamic optimization provides exact information on the sizes of intermediate results and hence can determine the optimal strategy. The last step is the execution of the produced access plan to answer the query.

Retrieving information from a collection of independently designed databases is a difficult task. Access to data requires planning to control the time required

to process a query. A user submits a global query which in the multidatabase case contains all the information necessary for retrieving local data. The query is decomposed into a set of subqueries - one for each DBMS that will be involved in query execution. Each of these subqueries may in turn be executed after further decomposition. The query optimizer creates an access strategy to evaluate the query. The number of strategies for executing a single query in a distributed system is far more than in a centralized system. The cost function to be minimized is the sum of the CPU, IO, and the communication costs. The communication cost is the time needed for shipping data between sites participating in the execution of the query. The cost function can also be the response time of the query in which case the parallel execution of operations is maximised. Then the access strategy is executed and the results combined to form a response to the original query. Initial query processing usually occurs at the site where the query is submitted, although some systems transmit the queries to designated servers for processing.

In the next chapter, we describe multidatabase systems in more detail. We also examine the different approaches in constructing a multidatabase system and the capabilities of each kind of system. In Chapter 3, we describe our multidatabase system and the facility which it uses for inter-database joins in a heterogeneous environment. Chapter 4 reviews some distributed query optimization algorithms and presents an algorithm for optimizing query execution in our system. Chapter 5 contains the conclusions of this work.

Appendix A contains our sample local and remote databases. Appendix B shows some example input queries and their response.

Chapter 2

Multidatabase systems

The use of different DBMSs has proliferated in recent times. This has resulted in a scenario wherein there exist databases which have to be interconnected to form a multidatabase system. The individual specifications have to be integrated in a bottom-up manner to generate a global specification. There are basically two approaches to this.

1. Global schema approach
2. Multidatabase language approach

2.1 Global schema approach

The global schema approach to multidatabases makes global access quite user-friendly [Landers82]. Global users see a single large, integrated database. The global interface is independent of all heterogeneity in local DBMSs and data representations. Global database administrators (DBAs) are required to design the global schema. They must have extensive knowledge of all the input schemata and user requirements of the global system to decide how to integrate the local schemata. Each of the local schemata is assumed to be optimized to local

requirements. An optimal design for global requirements can take into account local optimizations but cannot change them. Hence, the global DBA must also understand the local optimizations before deciding on the global schema.

2.1.1 Design of global schema

There are two stages of design, namely, schema integration and data integration [Dayal84].

Stage I: Schema integration

1. The first step is to identify a common data model for the global schema. The selected data model and query language should have the following properties:

   - They should allow a simple translation from the data models and query languages of the DBMSs constituting the heterogeneous system.
   - They should be suited to represent data and processing of component databases conveniently. In particular, the query language should possess 'set-oriented' primitives. This excludes the choice of complex data models and procedural languages.

   The relational model is the most widely used for this purpose, although variants of the entity-relationship model have been used in some instances. If a component schema is defined in terms of a different data model, it must be specified in the global schema data model. This step is called schema translation.

2. The similarities in component schemata should be recognized. Matches can be recognized because of structure similarities of overlapping portions

of schemata. Matches can be deduced from similarity of data in the pre-existing databases. All the matching portion of schemata can then be directly retained in the global schema.

3. Integration should identify conflicts in local schemata and resolve them. The types of conflicts are:

   - Naming conflicts - There are two types of naming conflicts. Synonyms occur when data objects which represent the same objects in the real world have been given different names in the two schemata. This is handled by giving same names to such entities. Homonyms occur when data objects with the same name correspond to different objects in the real world. In this case, the entities are given different names in the global schema. In both cases, name correspondence tables are stored as part of the definition of the global schema.
   - Scale differences - The same function values might be stored using different scales. The data should be retrieved using the more precise scale and the conversion formula stored as part of the global schema. In more complex situations, the conversion might require an arbitrary DBA-defined procedure.
   - Structural differences - These stem from different design choices for the local schemata. A real world object might be an entity in one schema and an attribute in another. Since a few structural differences can be handled by generalization, complex query modification procedures need to be written in most cases and these are stored as part of the global schema.
   - Domain differences - Here, some entities are not represented in one of the schemata. Detection of these differences is accomplished by

comparing source databases and noting inconsistencies. These should be stored in the global schema.

Apart from resolving these differences, it is also important to define any interdependencies between objects in different databases. Several methods have been suggested to aid in defining these interdependencies and algorithms have been proposed to ensure that the definitions are optimal and consistent. This also requires an extension of the capabilities of traditional DBMSs in the direction of more sophisticated multidatabase systems. For example, interactive tools have been developed to assist DBAs in schema integration [Sheth89]. Such tools collect the information required for integration, perform essential bookkeeping, and integrate schemata according to the semantics provided.

Stage II: Data integration

Once the structure of the global schema has been decided upon, its extension must be defined in terms of the extensions of the local schemata. But the extensions of the local schemata might present a conflict on the values of some functions. Several solutions have been proposed for this problem. One is to present all the conflicting data and the sources of each of them. Another is to present the most recent value. This method requires time-stamping of update operations. A third method is to present the value from the most reliable system. The assumption underlying this is that it is possible to determine the reliability of individual sites.

2.1.2 Maintenance of global schema

A global schema representing several database schemata can be produced by integrating two schemata initially and then integrating the rest one by one with the previous partial schema. Since a global schema can be a very large object, it may be difficult to replicate it at sites with minimum storage facilities. Some

systems solve this problem by replicating the global schema at specified server sites. This means that queries can be processed only at these sites, which may not be very convenient. Global DBAs must maintain the global schema in the face of changes to local schemata. The construction of the global schema should be such that any changes to local schemata should not significantly alter the global schema. However, addition or deletion of an entire database can cause enormous change. If the global schema is replicated at several sites, changes to all of them must be effected in as short a duration as possible. Otherwise, it can lead to inconsistent or invalid information.

2.2 Multidatabase language approach

The multidatabase language approach [Hurson91] is used to resolve some of the drawbacks of the global schema approach, such as the development time to create the global schema, large maintenance requirements, huge storage requirements, etc. In the global schema system, it is the responsibility of the global DBA to perform the integration. The DBA has total control over the data. A multidatabase language system transfers the integration responsibility to the user. To aid the user, it provides many functions and a great deal of control over the data. In this case there is no global schema. The user will see multiple autonomous schemata. So, a basic requirement for a multidatabase language is to define a common name space across all component schemata. A common name space can still provide some measure of location transparency if object names are independent of the site they reside at.

Most of the language extensions beyond standard database capabilities are involved with manipulating information representations. Queries to multiple databases, without a global schema, are called multidatabase queries. Since representation differences exist when the user submits such a query, the language

must have the capability of transforming source information into the representations required by the query. In this context, it is particularly desirable to make the multidatabase language non-procedural. The multidatabase system should also be capable of making good implicit decisions in interpreting what the user wants to accomplish and providing appropriate functions. The system will be easier to use if it can, by default, handle more complexity. The user may be working with multiple equivalent objects with slightly varying attributes. If the system can apply a single operation to all the data objects with consistent results, the user can give the objects a group name and invoke the operation with a single command. This is one of the important features of a multidatabase language. This can lead to support for implicit join capability wherein the user has to specify only the result format, and the system figures out which relations to join in order to produce the required result.

The multidatabase user need not have a knowledge of all the component schemata. He can acquire a knowledge of the databases dynamically as need arises. This is not a complex task in general because the user usually has some knowledge of the conceptual universe he is dealing with. The multidatabase language should provide functions to display what information is available from the various sources. This is particularly important bacause the sheer size of available data will make the task of finding necessary data difficult. The language also typically provides the ability to limit the scope of a query to the pertinent local databases.

2.2.1 Features of a multidatabase language

A multidatabase language typically provides features not found in traditional DBMSs [Litwin87]. Some of these features are discussed below.

Multiple queries are intended for situations where various databases model the

same universe. Then the user may wish to broadcast the same manipulation to several databases. Multiple queries are formulated through multiple identifiers or through options on the target list. A multiple identifier is a name shared by several attributes or relations. Consider, for example, the following query in the multidatabase system language (MDSL).

```
open M G
-range (y R)
-select y
-where (y.type = "chinese")
```

Here, the designator 'R' is the multiple identifier of the 'R' relations in both the M and G databases. The result of such a query is a set of two relations where each relation inherits the database name its attributes come from.

Options in the target list allow the user to specify attributes which may not exist in some of the schemata. Traditional DBMSs assume that all the attributes in the target list of a query are present in the schema of the addressed database. A list of the form $a_1|a_2|\ldots|a_n$ indicates that only one of the a_i's should be chosen. The choice follows the list order. Thus, if a_1 is an attribute in schema 1 and a_2 is an attribute in schema 2, the target list selects the corresponding attributes in both the schemata. An attribute of the form '~a' indicates that 'a' is optional.

It is possible to define aliases and abbreviations for databases names. This helps in resolving name differences between databases. A name is allowed to refer to multiple objects from different databases if the objects are semantically equivalent. This implies that an operation on a named object may actually cause multiple operations on several databases to occur.

MDSL allows the user to define dynamic attributes. A dynamic attribute

is a temporary attribute defined by a mapping from existing attributes. It is dynamically defined in a query and is unknown to any schema. It is used to accomplish transformations in data formats. It can also be used to abstract attribute values from multiple sources into a single set of values. However, representation transformations need not be performed for every query, so the dynamic attribute need not be accessible across queries.

A multidatabase language should have the ability to present results in a variety of formats. When multiple data sources are used, a user may wish to see all the results produced by each source. Also, the user may want the system to "calculate" a result from a set of results and present this value. In either case, the system should have the ability to present values in a format different from the one in which they are stored.

Another important feature of such a language is to facilitate inter-database joins. This is necessary when information in several databases are to be related. To select data from several databases with a single query, the target list should contain attributes from the different databases. This feature is an essential part of any multidatabase query capability.

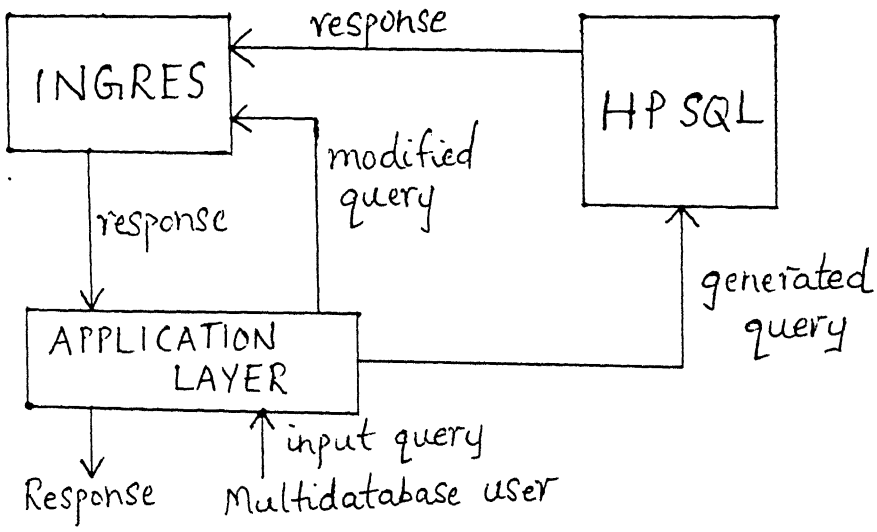
In the following part of this report, we discuss the design and implementation of a facility for inter-database joins in an existing query language for a heterogeneous system. Once a query which involves inter-database joins is submitted to the system, it has to be processed efficiently. We show how such a query can be processed in our multidatabase system.

Chapter 3

An experimental multidatabase system

Our multidatabase system consists of two databases. Both the databases are at the same site. In this sense, they are logically distributed databases. Both the databases use the relational model. but are heterogeneous in nature. One of the databases uses an INGRES database management system while the other runs on a HP SQL database management system. The INGRES system supports updata capability. This means that the user can create tables in the INGRES database as well as query them. On the other hand, the HP SQL system supports only retrieve capability. Hence, data can only be retrieved from the HP SQL database. The multidatabase user submits a global query to the INGRES system for processing. This query can contain references to data residing in HP SQL tables. We provide a facility for distributed processing wherein data at different sites in a heterogeneous environment can be brought together at the query site in order to answer a query. In our case, the query site is the INGRES system. Since the user does not have direct access to the HP SQL system, we provide an application layer above the INGRES system which issues commands to query the HP SQL tables. The data thus retrieved is inserted into specially

created tables in the INGRES system. The figure below shows the architecture of our system.



We present a brief overview of both the DBMSs.

3.1 Overview of INGRES

INGRES is a relational database management system which provides an extensive set of built-in functions for assisting in the process of developing database application [Date87]. The major subsystems in INGRES are:

- INGRES/QUERY - database retrieval/update, data entry
- INGRES/REPORTS - report definition and report writing
- INGRES/GRAPHICS - business graphics
- INGRES/FORMS - form definition and editing
- INGRES/APPLICATIONS - application generation

All these subsystems constitute the INGRES frontends. INGRES frontends are form-based. A form is a display-screen version of a familiar paper form which can be used for both input and output. The process of interacting with a forms-based system or application is referred to in INGRES as visual programming. These forms-based systems provide facilities for data entry and maintenance and for building customized applications.

The INGRES backend provides all the basic DBMS functions and handles query processing in the INGRES system. INGRES provides two query languages to the user - QUEL and SQL. These languages provide a high-level of abstraction which increases the user productivity. A query language user always operates within the context of a single database. Hence, the system does not allow a user to manipulate data in a non-INGRES database at a remote site. Often, a group of SQL statements are dispatched to INGRES for processing in an atomic fashion. Either all the statements are processed as a group or none at all. Such a group of statements is called a transaction. Transaction processing ensures database integrity. The savepoint feature ensures that if any errors occur during processing, the correction is limited to those erroneous parts of the database. The rollback feature is responsible for undoing the changes to the database.

INGRES provides several system-level commands which operate at the level of the machine's operating system. Terminal Monitors are provided for each of the query languages which support interactive access to an INGRES database via its respective language. INGRES provides preprocessors for several host languages. There are commands which deal with various aspects of performance, optimization, and recovery. Each INGRES system supports one special database called the database database (DBDB), together with several user databases. The DBDB is used for control purposes. The INGRES system provides utilities which operate on the DBDB. These can be used only by the system manager.

INGRES provides four levels of support for distributed processing. They allow several INGRES sites to be connected together in a network. They also allow an arbitrary collection of tables from the local databases to be defined as a distributed database for data manipulation purposes.

The query language of INGRES can be used interactively and as a database programming language. This means that SQL statements can be embedded in a host language program to perform database operations.

To sum up, the advantage of INGRES is simplicity which translates into easy usability and productivity. Other salient features are high-level languages which can be used in dual mode, forms-based subsystems for ease of application development, dynamic data definition facilities, and the ability to operate in several environments.

3.2 Overview of HP SQL

HP SQL is a relational database management system which allows a user to create relational databases and write application programs to serve them. It provides the basic DBMS functions and few extensions beyond it. The query language provided by the system is SQL. Each HP SQL database is associated with a DBEnvironment. Before a SQL statement can access a HP SQL database, the user has to identify the environment in which accesses will be performed. SQL can be used interactively or embedded in a host language (such as C). An embedded SQL program has to be preprocessed before it can be executed against the database. During preprocessing, the preprocessor needs to access the same DBEnvironment which the source code will access at runtime. To access more than one HP SQL DBEnvironment from the same program, the program must be separated into different compilable sections. Each section must access only one DBEnvironment, and must be preprocessed and compiled separately.

Finally all the compiled modules can be linked together to create an executable program. HP SQL authorization governs who can maintain a program that accesses a DBEnvironment.

HP SQL has the concept of sections. A section consists of HP SQL instructions for executing an SQL command. A section serves the purposes of access validation and access optimization. By creating and storing sections at processing time rather than at runtime, performance is improved.

HP SQL allows a user to manipulate databases in several native languages in addition to the default language. If the proper message files have been installed, HP SQL displays prompts, messages, banners, etc. in the selected language, and displays dates and times according to local customs. It also accepts responses in the selected native language.

The system does not provide any support for distributed processing. In this context, it supports centralized database management. Also, no special system utilities or interfaces are provided.

SQL commands that are not known until runtime can also be processed by an application program. Such commands, called dynamic commands, are entered by the user at runtime. The commands are passed onto the system in a character array variable. HP SQL converts these commands into executable instructions at runtime. These instructions are deleted at the end of the associated transaction.

In short, HP SQL is a simple database management system which can be used for managing databases of relatively small sizes.

3.3 Inter-database joins

We describe a facility for inter-database joins in our multidatabase system which we have described above. The access languages of the two systems are different

in their functionality. We select the query language of INGRES, SQL, as the language of our system. This means that the user submits a query on the INGRES system. The user does not have direct access to the HP SQL database, except possibly through applications which serve the HP SQL database exclusively. We refer to the INGRES database as the local database and the HP SQL database as the remote database.

SQL provides the SELECT statement to retrieve data from a database. The multidatabase user may want to manipulate data from the local database only or data jointly from the remote database. To specify data from the remote database, the database name is used as a prefix for unique identification of data objects. This implies that database names are unique in our system which is indeed the case. The local database name need not be specified in the query because, by default, all data objects referenced in the query are considered to be from the local database. To identify the database name to the SQL parser, we prefix the database name with a '$' sign. Thus, the way to access an attribute in the remote database is to have a reference of the form

```
$DB-NAME.RELN-NAME.ATTR-NAME
```

where DB-NAME is the name of the remote database. Prefixing the database name also accounts for the case where relations in different databases have the same name. This is not unusual in a multidatabase environment, especially if the databases model the same universe.

This feature provides the ability to perform inter-database joins. A join operation allows the user to compose two relations in a generalised way. We normally specify the joining attributes in the WHERE clause of the SELECT SQL statement. A clause specifying an inter-database join will be of the form

```
$DB-NAME1.RELN-NAME1.ATTR-NAME1 <operator>
        $DB-NAME2.RELN-NAME2.ATTR-NAME2
```

In our multidatabase system, we have only equijoins which means that the operator above is '='. If one of the joining attributes is from a local relation, the form of the clause will be

```
$DB-NAME.RELN-NAME1.ATTR-NAME1 = RELN-NAME2.ATTR-NAME2
```

Since a query submitted by the user may contain references to the remote database, it is first input to a query preprocessor. If the query contains only local references, it is output without any modification so that it can be executed by the local SQL processor. If the query contains references to the remote database, the local system has to access the data in the remote database. We solve this problem by downloading the remote database data to the local database. This is a popular approach adopted in many heterogeneous environments and also in workstation/server environments. Specialized programs have been developed which download portions of a database into a workspace and load it in a target database in an identical manner, retaining all indexing information. This process of downloading creates a temporary relation in the local database which needs to exist till the execution of the query is completed and the final answer presented to the user. The original relation name is retained for the temporary table. Since it is now a local table, the database name need not be prefixed. To indicate that it is a temporary relation, we prefix the relation name with a non-standard prefix which, in our case, is the sequence of alphabets 'ZZ'. Hence, a reference to a remote relation is transformed to a reference to a local relation in the user query. For example, a clause of the form

```
$DB-NAME.RELN-NAME1.ATTR-NAME1 = RELN-NAME2.ATTR-NAME2
```

will, after transformation, be of the form

```
ZZRELN-NAME1.ATTR-NAME1 = RELN-NAME2.ATTR-NAME2
```

The query preprocessor replaces all references to the remote database in the input query with references to corresponding temporary relations in the local database. This modified query is input to the local SQL processor for execution.

Before the query can be executed, the data from the remote database has to be transferred to the local database. Connection procedures, and transfer of data are transparent to the user. The simplest technique is to move all the data from a remote table to its corresponding local table. This is not acceptable because large amounts of data may have to be transferred and large amounts of temporarystorage might be required at the receiving site. Also, all the data in a table might not be necessary in answering a query. For example, in a relation which appears in a join clause, only those tuples which satisfy the join condition are candidates for further processing. In moving data from the remote database, it is particularly useful to identify the data which is not necessary so that it need not be shipped. Query processing techniques exist which aid the user in identifying the necessary portions of data. We use these techniques to develop a procedure which reduces the amount of data transferred between databases. The procedure and its implementation form the topic of the next chapter.

Chapter 4

Query processing

Query processing refers to the process of executing a query which results in a response to the query. Efficient processing of a query implies that a cost criterion must be satisfied while answering the query. A number of cost criteria can be used as objectives of query processing optimization, especially in a distributed environment [Hevner87]. Some of them are to reduce the number of data transmissions, to reduce the amount of data transferred, to minimize the response time to the user, to reduce the total resources used by the query, to improve system throughput, to equalize the workload at all sites, etc. In our multidatabase system, we use a strategy which reduces the amount of data transferred between the databases.

We have mentioned in the previous chapter that a query containing remote references is modified to a query containing only local references. In the process, temporary relations are created in the local system which contains data from the remote database. Te retrieve the remote data, a query has to be executed on the remote database. Typically, for a query executed on the remote database, a temporary table is created at the local site and the data retrieved by the query is loaded into the newly created table. This means that the original query has to be transformed into a set of queries of which one is the modified input query to

be executed locally, and the rest are 'generated' queries to be executed remotely. In a general situation, if N is the number of remote relations referenced in an input query, N queries are executed at the remote site. But in some cases, M < N queries may suffice to move all the required data to the local database. We first identify the relations to be moved to the local database. We next present an algorithm which performs a reduction on the size of these relations before they are shipped. The SQL query which we consider is of the form

```
Select  T(S)
From    F(S)
Where   Q(S)
```

Here, $T(S)$ is the target list which represents the list of attributes being selected by the query.

$F(S)$ is the list of relation names from which the attributes are selected.

$Q(S)$ is the qualification of the query. We restict Q to be a list of predicates which are in conjunction with each other. Typically, Q is of the form

$$P_1 \text{ AND } P_2 \text{ AND } \ldots \text{ AND } P_n$$

Queries can be represented in several forms for ease of processing. A chosen form should be powerful enough to express a large class of queries and it should provide a well-defined basis for query transformation. Relational calculus and relational algebra are well-known representation forms. Another representation form for queries are graphs. The visual representation of a query contributes to an easier understanding of its structural characteristics. In addition, graph theory offers several results useful in the analysis of graphs. We present two graph models of a query [Bernstein81], viz. join graphs and query graphs.

4.1 Join graphs

For a query S with qualification Q, we define the join graph to be a node-labelled undirected graph $J(V, E)$ where

$$V = \{R_i.A \mid A \text{ is a domain of } R_i\}$$

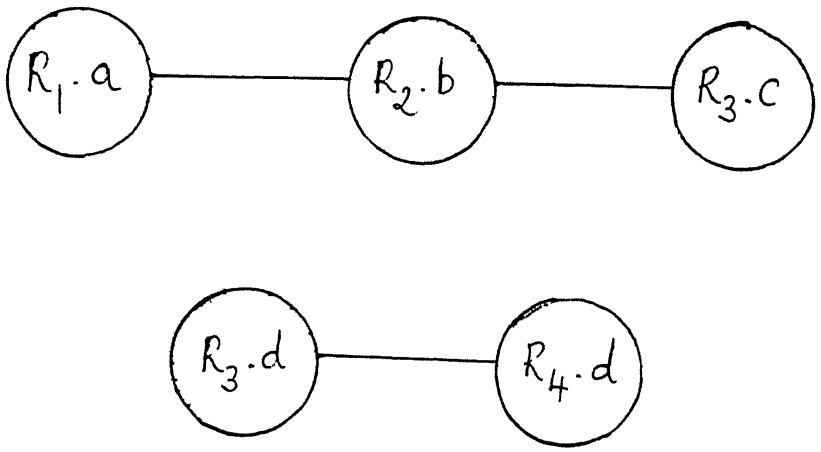
and

$$E = \{(R_i.A, R_j.B) \mid (R_i.A = R_j.B) \text{ is a join clause in Q}\}$$

For example, the join graph for the qualification

$$(R_1.a = R_2.b) \text{ AND } (R_3.d = R_4.d) \text{ AND } (R_2.b = R_3.c)$$

is shown below.



The join graph simply represents the joins in the qualification by edges in the graph. A join graph is total when it contains all possible edges between vertices. A join graph is reduced when some of the edges between some vertices are missing. Two types of reduced join graphs are particularly important [Ceri84].

1. A reduced graph is partitioned if the graph is composed of two or more

components (subgraphs).

2. A reduced join graph is simple if it is partitioned and each component has just one edge.

A simple join graph is important in query optimization. A pair of relations which are connected by an edge in a simple join graph have a common set of values of join attributes. If both the relations reside at the same site, the join clause can be employed to restrict the number of tuples for further processing.

4.2 Query graphs

For a query S with qualification Q, we define the query graph $G(V_s, E_s)$ as

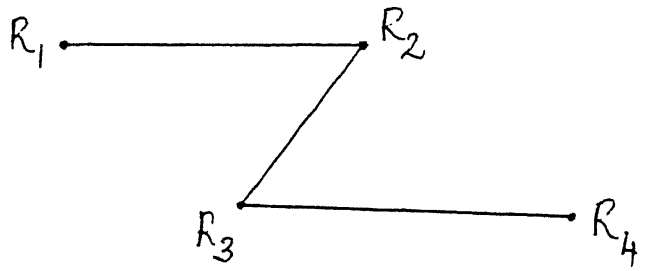
V_s = set of all relations referenced by Q

E_s = $\{(R_i, R_j) \mid i \neq j$ and a clause of Q references relations R_i and $R_j\}$

Since more than one join is possible between two relations, G is a multigraph in general. For example, the query graph for a query with the following qualification

$$(R_1.a = R_2.b) \text{ AND } (R_3.d = R_4.d) \text{ AND } (R_2.b = R_3.c)$$

is as shown below.

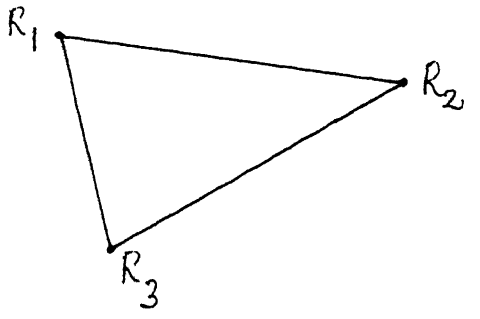


Several semantically equivalent expressions may exist for a given query. One source of difference between two equivalent expressions is their degree of redundancy. A straight-forward evaluation of a redundant expression would lead to the execution of a set of operations some of which may be superfluous. Therefore, query optimization aims at the elimination of redundancy by means of transforminga redundant expression into an equivalent non-redundant one.

Consider a query S whose qualification is given by

$$(R_1.A = R_2.B) \text{ AND } (R_2.B = R_3.C) \text{ AND } (R_3.C = R_1.A)$$

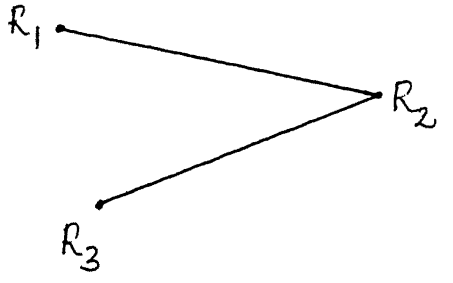
The query graph for this query shows a cycle.



We note that we can drop any of the clauses in the qualification without altering the semantics of the query. Removing the last clause, for example, results in the qualification

$$(R_1.A = R_2.B) \text{ AND } (R_2.B = R_3.C)$$

which has the following query graph.



This graph is acyclic in nature. The property of the cycle that permits us to drop one clause is that each relation participates in the cycle with only one joining domain. Consequently, we classify queries into two groups. A query is a tree query if its query graph is a tree or if it is equivalent to a query whose query graph is a tree. All other queries are cyclic queries.

A reducer is an operation that can be applied to reduce the cardinality of its operands. A full reducer fully reduces its operands in the sense that it outputs only those tuples which are required to answer a query. It is useful to identify when full reduction is possible for a query. Query graphs help in this identification process.

4.3 Optimization algorithms

The standard approach for evaluating queries is to form the join of the relations involved, apply the restrictions to the intermediate result, and finally project it onto the attributes in the target list. This is difficult if the relations reside at different sites. Several algorithms have been suggested for such a distributed environment.

1. Hill-climbing algorithm

   This algorithm is quite general and can be applied to meet different objectives [RothnieS0]. It computes progressive refinements of an initial feasible solution until no beneficial modifications of the evaluation plan are possible. The initial feasible solution is the minimum cost strategy among all the ones that transmit all the required relations to one site where the complete query is processed. This site is usually chosen as the site with the maximum amount of data.

2. Semijoin algorithm

This algorithm assumes that the communication costs dominate local processing costs [Sacco82]. Hence, relations are reduced as much as possible before shipping them to the target database. The semijoin of R and S takes the join of two relations R and S and projects back on the domain of relation R. That is, it retrieves those tuples in R that join with some tuple in S. In a sense, it is a generalization of restriction; it restricts R by values that appear in the join domain of S. Before transmission, a projection is locally performed to discard unnecessary attributes. The advantages of semijoin over the join operation are:

- Its evaluation only requires transmission of value lists of joining attributes instead of that of an entire relation.
- It has a reductive effect since the result of semijoin is always a subset of the relation involved, whereas a join may produce a Cartesian product in the worst case.

The semijoin algorithm consists of the following steps. Let A represent the join domains of R and S.

(a) Send $\Pi_A(R)$ to the site of S.

(b) Locally execute the join of $\Pi_A(R)$ and S at this site resulting in relation W.

(c) Send W to the site of R.

(d) Locally execute the join of W and R at this site.

Semijoins are important for hardware devices tailored for relational query processing because they provide a hardware semijoin instruction. The central problem in global reduction using the semijoin algorithm is finding

the best sequence of semijoins [Templeton87]. The cost of semijoining R with relation S is assumed to be a function of the size of the joining atttribute in S which must be copied to R's site. Since duplicate values are removed in the course of projection, the cost is estimated from the expected number of unique values of the joining attribute. The benefit is the amount by which R is reduced as a result of semijoin. Benefit is estimated from the total size of R and the selectivity of the joining attribute. The selectivity of an attribute is based on whether the attribute can assume few or many values. Semijoins are performed until no semijoin is found whose benefit is greater than its cost.

3. Replicate algorithm

This algorithm assumes that local processing costs dominate transmission costs [Sacco82]. Hence, this does not reduce relations but seeks to execute the user's query at several sites in parallel. It first finds an optimal set of sites at which the query can be executed. All sites containing relations referenced by the query are potential processing sites. The sites should be such that there will be complete copies of all other referenced relations at each selected site. Since no set of sites may meet this requirement, relations may have to be copied between sites. The chosen set is the one which requires least data transfer. After necessary data is moved between sites, the user's query is sent to each of these sites and is executed in parallel to provide a set of partial answers. The partial answers are moved to the result site where they are unioned into a temporary relation. This relation may constitute the final answer or it may require further processing to generate the final answer. Hence, the main features of this strategy are that it enhances parallelism, and permits the local DBMS to perform more local optimization. The strategy is more efficient if there is

a greater degree of replication of data.

4.4 A join-based algorithm

When local processing costs are considered along with transmission costs, it is advantageous to perform joins instead of semijoins [Ceri84]. Semijoins incur an extra join operation which, in some cases, might be expensive. Joins perform better under the following conditions.

1. There are few attributes in the target list of the query which do not appear also as join attributes of some join.

2. Semijoins have a low selectivity.

If an inter-database join is specified in a query, both the relations involved in the join need to be present at the same site before the join operation can be preformed. This means that either both the relations are moved to a third site or one of the relations is moved to the location of the other relation. In our system, data from a relation in the remote database is moved into a 'corresponding' local table and the join operation required by the query is performed at the local database. Thus, the objective of our query processing algorithm is to reduce the amount of data transferred to tables in the local database.

4.4.1 The algorithm

Given an input query S, our algorithm transforms it into a modified query L in which a reference to a remote relation P is replaced by a reference to the corresponding local table P'. During this process, it generates a sequence of queries to be executed at the remote database. Each of these queries R transfers data from a remote relation to a local relation.

Notation

For a query S, the target list is represented by T(S) and the qualification by Q(S). RT(S) represents the set of remote attributes in the target list. P denotes a local relation while #P denotes a remote relation. The input query is

```
S :  Select T(S)
     From   A, B, ..., #M, #N, ...
     Where  Q(S)
```

After preprocessing, the modified query L has the form

```
L :  Select T(L)
     From   A, B, ..., M', N', ...
     Where  Q(L)
```

The remote query R has the form

```
R :  Select T(R)
     From   #M, #N, ...
     Where  Q(R)
```

We now present our algorithm.

1. Q(L) — Q(R) — ∪
2. Represent the qualification of S by a join graph J_s.
3. for each connected component C of J_s do

   If C does not reference remote relations, add the clause(s) represented by it to Q(L). Delete C from J_s.

If C does not reference local relations, add the clause(s) represented by it to Q(R).

Partition the nodes of C according to the relations named by them.

Connect all nodes representing a remote relation through a chain. Denote this group of nodes by C(R).

Connect all nodes representing a local relation through a chain. Denote this group of nodes by C(L).

Add the clause(s) represented by C(R) to Q(R).

Add the clause(s) represented by C(L) to Q(L).

If C(L) = ϕ, add the clause(s) represented by C(R) to Q(L).

Let local-attr be any element of C(L), the set of local attributes in C.

Let K = RT(S) $\cap$ C(R).

If K = ϕ, let remote-attr be any element of C(R)

else let remote-attr be any element of K.

If C(L) = ϕ, Q(L) = (local-attr = remote-attr)

else Q(L) = Q(L) AND (local-attr = remote-attr)

RT(S) = RT(S) $\cup$ {remote-attr}

endfor

4. T(L) = T(S), the target list of given query.

Group all attributes in RT(S) according to the relation to which each attribute belongs. For each group, we have a query which is executed remotely. All the attributes of a group form the target list of its respective query. Each such generated query has the same form, with the target lists being different in each case. For a group identified by relation #P, add

relation #P to the FROM clause of the local query L. Since we change the name of the transferred relation, replace each occurrence of #P in L by its name-transformed equivalent P'. P' refers to the local relation which contains data from the remote relation #P. ■

Correctness of algorithm

The algorithm reduces a remote relation by applying as many restrictions and projections as are applicable at the remote database. Projections restrict the relation to the required attribute lists only. Join clauses further reduce the relation by eliminating tuples which do not satisfy at least one join clause of the qualification. We also use two other techniques for reduction.

1. Constant propagation This method propagates a constant across join clauses [Gardarin89]. For instance, the qualification

   (#M.a = B.a) AND (B.a = 10) AND (#M.a = C.b)

   is equivalent to the qualification

   (#M.a = B.a) AND (#M.a = C.b) AND (#M.a = 10)

   In this case, the restriction #M.a = 10 can be applied at the remote database.

2. A join clause can be logically implied by the qualification by transitivity laws. For example,

   (A.a = #M.a) AND (A.a = #N.a) imply (#M.a = #N.a)

   which, clearly, is applicable at the remote database.

Since the above techniques can be easily seen to be correct, and our algorithm employs only these techniques, our algorithm correctly performs a reduction on a remote relation.

Q(L) is obtained by the following method. All join clauses in the input query which refer to a local table or a remote table only form a part of Q(L) directly. In the case of constant propagation, the join clause applied at the remote database is simply excluded from the qualification of the modified query. For example, the qualification

(#M.a = B.a) AND (B.a = 10) AND (#M.a = C.b)

is equivalent to the qualification

(#M.a = B.a) AND (#M.a = C.b) AND (#M.a = 10)

as discussed above. The last clause is detached, thus giving the qualification

(#M.a = B.a) AND (#M.a = C.b)

This qualification correctly retrieves only those tuples which would have been selected by the original qualification.

In the final case, if the qualification has a cycle as in

(#M.a = A.a) AND (A.a = B.a) AND (B.a = #M.a)

we can drop any of the clauses; the resulting qualification is obviously equivalent to the above one. Also, qualifications having the same transitive closure are equivalent. Using these two properties and the fact that a clause which is part of a cycle and is applied at the remote database need not be applied twice, we obtain Q(L) for the modified query.

To illustrate, consider the qualification

(A.a = #M.a) AND (A.a = #N.a)

The transitive closure is given by

(A.a = #M.a) AND (A.a = #N.a) AND (#M.a = #N.a)

We drop the first clause, giving

(A.a = #N.a) AND (#M.a = #N.a)

The second of these clauses can be applied at the remote database and hence can be detached, giving

(A.a = #N.a)

as the qualification for the modified query. It can be readily seen that the qualification thus obtained is correct.

Hence, the algorithm correctly produces the sequence of queries R and L.

4.4.2 Examples

Example 1

Consider the query S given by

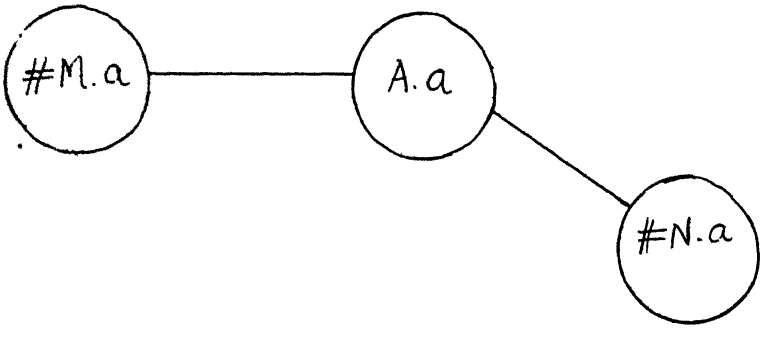
```
S : Select A.b
    From A, #M, #N
    Where (A.a = #M.a) AND (A.a = #N.a)
```

T(S) = {A.b}

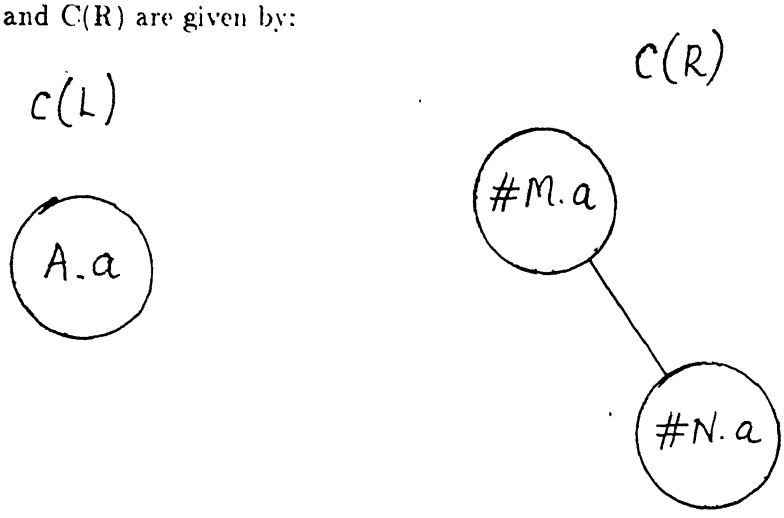
RT(S) = ϕ

Q(S) = (A.a = #M.a) AND (A.a = #N.a)

The join graph for Q(S) is as shown below.



C(L) and C(R) are given by:



Hence, Q(R) = (#M.a = #N.a)

local-attr = A.a

remote-attr = #M.a

Q(L) = (A.a = #M.a)

RT(S) = {#M.a}

There is one remote query R given by:

```
R : Select #M.a
    From #M, #N
    Where #M.a = #N.a
```

The modified local query is

```
L : Select A.b
    From A, M'
    Where  (A.a = M'.a)
```

Example 2

Consider the query S given by

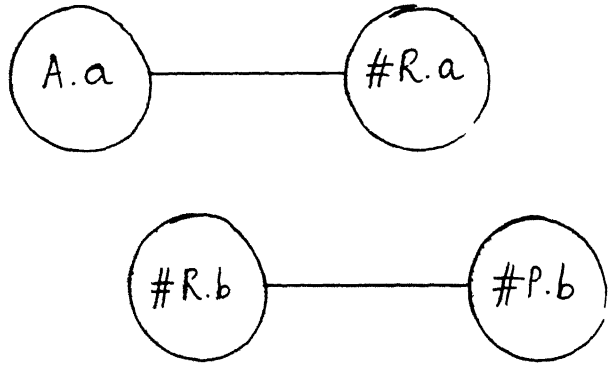
```
S : Select #P.a, #R.d
    From  A, #P, #R
    Where (A.a = #R.a) AND (#R.b = #P.b)
```

We have T(S) = {#P.a, #R.d}

RT(S) = {#P.a, #R.d}

Q(S) = (A.a = #R.a) AND (#R.b = #P.b)

The join graph is as shown below.



For the first component of the graph, C(L) and C(R) are as shown.

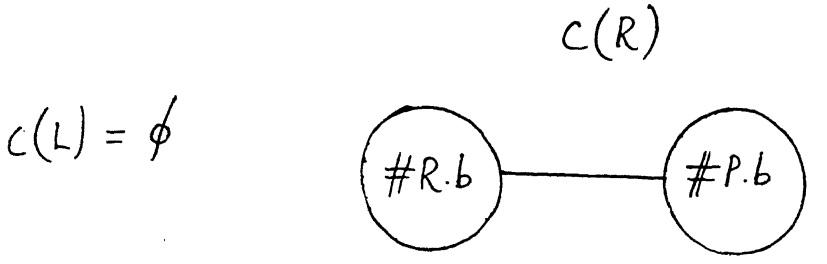
By the algorithm, local-attr = A.a

remote-attr = #R.a

Q(L) = (A.a = #R.a)

RT(S) = RT(S) ∪ {#R.a}

For the second component, C(L) and C(R) are as shown.



Q(R) = (#R.b = #P.b)

Q(L) = Q(L) AND (#R.b = #P.b)

RT(S) = RT(S) ∪ {#R.b, #P.b}

= {#P.a, #R.d, #R.a, #R.b, #P.b}

There are two remote queries.

```
R1 : Select #P.a, #P.b
     From #P, #R
     Where (#R.b = #P.b)
```

```
R2 : Select #R.a, #R.b, #R.d
     From #P, #R
     Where (#R.b = #P.b)
```

The modified local query is

```
L :  Select P'.a, R'.d
     From A, P', R'
     Where (A.a = R'.a) AND (R'.b = P'.b)
```

4.5 Implementation procedure

The program takes a user query as input and produces the modified query as output. The modified query is executed to yield the final response. The

program has been developed in the C language. Several data structures are used to represent the structure of the input query.

Information about each of the remote relations is stored in a structure which has got four component fields. The C structure definition is as follows:

```
struct aliastable {
        char tablename[30];
        char localname[30];
        char aliasname[20];
        char colnames[100];
        struct aliastable *next;
        };
```

A description of each of these fields is given below.

- tablename - holds the name of the remote table
- localname - holds the name of the corresponding temporary table in the local database
- aliasname - holds any alias used by the query for the relation
- colnames - represents the names of columns which occur in the query

One structure holds the information about one relation. All the structures are chained together to form a linked list.

There is a global pointer to the aliastable, which initially is NULL. This pointer has the definition

```
struct aliastable *aliasptr;
```

The functions aliasalloc( ) and freestorage( ) allocate and de-allocate storage, respectively, for an instance of the structure.

We use two arrays named larray (for local array) and rarray (for remote array) to represent the join clauses of the query. The two arrays relate to each other through their corresponding elements. The clause

(A.a = #M.b)

is represented by the assignment statements

larray[i] = A.a and rarray[i] = #M.b

for some index i. This method is versatile enough to represent any kind of join clause that may appear in a query. We extend each element of the two arrays to hold multiple names, separated by a blank. Then, for any index i, the elements of larray[i] and rarray[i] taken together represent all the join clauses, implied or otherwise, present in the input query. The definitions of the two arrays are:

```
char larray[10][100];
char rarray[10][100];
```

Thus, for example, the elements

larray[3] = 'A.a B.a' and rarray[3] = '#M.a'

together represent the set of join clauses

(A.a = B.a) AND (B.a = #M.a)

When a 'generated' query is to be executed, a call is made to a procedure. This procedure first creates a temporary table in the local database, whose name is given by the 'localname' field in the structure for the relation in question. It then queries the remote table and retrieves data from it. This data is loaded into the newly created relation.

Chapter 5

Conclusions

The join-based algorithm which we have presented in the previous chapter performs well when local processing costs are comparable to transmission costs. This algorithm can be easily extended to include multiple databases at several sites. In a heterogeneous environment, not all the DBMSs provide the same functionality to the multidatabase user. In such a case, the algorithm can take into consideration the type of capability provided by each of the local systems. The algorithm can also be used when there is replication of data at different sites. In this case, a copy of each relation at a particular site is treated as a local relation, if the site is the query site, and as a remote relation otherwise. The requirement of the algorithm is that the DBMS at the query site should provide update capability.

The algorithm forms the basis of a feature for inter-database joins in a heterogeneous scenario. Existing query languages do not provide such a facility. This facility is important in a multidatabase system, as is evidenced by a growing interest in multidatabase languages. DBMSs should also provide support for temporary tables. It must provide facilities to dynamically define the structure of temporary tables, and to manipulate them without altering the state of the database. The temporary relations should be accessible by DBMSs at other

sites. This will enable a user to tailor the multidatabase system to his specific needs.

Further extension to this work lies in the area of providing additional data manipulation capability for remote data. For example, update capability can be provided for remote tables. With this capability, alternative techniques for query optimization in multidatabase systems can be suggested. For instance, a semijoin-based algorithm may be more effective in systems where queries are not ad-hoc in nature. In a more general situation, query processing can involve identifying sites with update capability and modifying the input query accordingly.

Bibliography

[Bernstein81] Bernstein, P.A. and D.W.Chiu. "Using Semi-Joins to Solve Relational Queries." *JACM*, Vol.28, No.1, 1981, pp.25-40.

[Ceri87] Ceri, S. et al, "Distributed Database Design Methodologies," *Procs.IEEE*, Vol.75, No.5, 1987, pp.533-546.

[Ceri84] Ceri, S. and G.Pelagatti, *Distributed Databases: Principles and Systems*, McGraw-Hill, 1984.

[Date87] Date, C.J., *A Guide To Ingres*, Addison-Wesley, 1987.

[Dayal84] Dayal, U. and H.Hwang, "View Definition and Generalization for Database Integration in a Multidatabase System," *IEEE Trans. Soft. Engg*, Vol.10, No.6, 1984, pp.628-645.

[Epstein78] Epstein, R. et al, "Distributed Query Processing in a Relational Database System," in *ACM SIGMOD*, 1978, pp.169-185.

[Gardarin89] Gardarin, G. and P.Valduriez, *Relational Databases and Knowledge Bases*, Addison-Wesley, 1989.

[Hevner87] Hevner, A.R. and S.B.Yao, "Querying Distributed Databases on Local Area Networks," *Procs.IEEE*, Vol.75, No.5, 1987, pp.563-571.

[Hurson91] Hurson, A.R. and M.W.Bright, "Multidatabase systems: An advanced concept in handling distributed data," in *Advances in Computers*, Vol.32, 1991, pp.149-200.

[Landers82] Landers, T. and R.L.Rosenberg, "An Overview of Multibase," *Procs.2nd Int. Symp. on Distributed Databases*, 1982, pp.153-183.

[Litwin87] Litwin, W. and A.Abdellatif, "An Overview of the Multi-database Manipulation Language MDSL," *Procs.IEEE*, Vol.75, No.5, 1987, pp.621-632.

[Rothnie80] Rothnie, J.B. et al, "Introduction to a system for distributed databases (SDD-1)," *ACM TODS* 5(1), 1980, pp.1-17.

[Sacco82] Sacco, G.M. and S.B.Yao, "Query optimization in distributed database systems," in *Advances in Computers*, Vol.21, 1982, pp.225-273.

[Sheth89] Sheth, A.P. and J.A.Larson, "A Tool for Integrating Conceptual Schemas and User Views," *Procs.5th Int. Conf. on Data Engg*, 1989, pp.176-183.

[Templeton87] Templeton, M. et al, "Mermaid - A Front-End to Distributed Heterogeneous Databases," *Procs.IEEE*, Vol.75, No.5, 1987, pp.695-708.

[Ullman80] Ullman, J.D., *Principles of Database Systems*, Computer Science Press, 1980.

[Yu84] Yu, C. and C.Chang, "Distributed Query Processing," *ACM Comput. Surv*, Vol.16, 1984, pp.399-433.

Appendix A

Example local and remote databases

Remote relation : Reference

Title	Author	Courseno
Databases	YYY	CS 700
Concrete Structures	NNN	CE 749
AI and Speech	JJJ	CS 750
Knowledge-based Systems	RRR	CS 600
Databases	RRR	CS 500

Local relation: Books

Accno	Callno	Title	Author	Field
A10850	010.500	YYY	Databases	Databases
A62580	025.894	ZZZ	Structures	Structures
A528758	011.892	HHH	Fluid Mechanics	FM
A52950	010.855	JJJ	AI and Speech	AI
A51050	010.966	JJJ	Artificial Intelligence	AI
A209889	011.500	RRR	Introduction to Databases	Databases
A10585	043.443	NNN	Concrete Structures	Structures

Note: Read 'Title' field as 'Author' and vice-versa.

Local relation: Issue

Accno	Idno
A10850	202
A62580	205
A51050	204
A52950	9101143
A528758	9150498
A209889	9150498

Remote relation : Students

Name	Rollno	Dept	Programme
AAA	9100000	AE	M.Tech
BBB	9110112	CS	M.Tech
CCC	9101143	CS	Ph.D
DDD	9140549	CE	M.Tech
EEE	9150498	ME	Ph.D
FFF	9101485	CS	M.Tech
GGG	9111295	CE	M.Tech

Remote relation: Faculty

Name	PFNo	Dept
PPP	200	AE
QQQ	201	CS
RRR	202	CS
SSS	203	ME
TTT	204	CS
UUU	205	CE
VVV	206	AE

Remote relation: Courses

Courseno	Coursename
CS 700	Fault Tolerance in Databases
AE 655	Fluid Flows
CE 749	Design of Concrete Structures
CS 750	AI & Speech Recognition
CS 500	Database Interfaces
CS 600	Knowledge-based Systems

Remote relation: Enrol

Rollno	Courseno
9110112	CS 750
9101143	CS 750
9140549	CE 749
9100000	AE 655
9150498	CS 500
9101485	CS 750
9111295	CE 749

Remote relation: Offer

Name	Courseno
PPP	AE 655
RRR	CS 700
TTT	CS 750
UUU	CE 749
RRR	CS 500
QQQ	CS 600

Appendix B

Example queries

1. Query : List the names of courses for which a reference book has been issued to a student enrolled in the course.

   SQL formulation of the above query:

```
SELECT C.COURSENAME
FROM BOOKS B, REFERENCE R, ISSUE,
     $STUDDBE.ENROL E, $STUDDBE.COURSES C
WHERE E.COURSENO = C.COURSENO AND
C.COURSENO =R.COURSENO AND R.TITLE= B.TITLE AND
R.AUTHOR=B.AUTHOR AND B.ACCNO=ISSUE.ACCNO AND
ISSUE.IDNO = E.ROLLNO;
```

   Answer:

   AI & Speech Recognition

2. Query : List the titles of books issued to CS students.

   An embedded SQL program for the above query follows.

```
#include <stdio.h>

EXEC SQL INCLUDE SQLCA;

#define ok 0

main()
{
    EXEC SQL BEGIN DECLARE SECTION;
    char title[20];
    EXEC SQL END DECLARE SECTION;

     EXEC SQL WHENEVER SQLERROR STOP;
     EXEC SQL CONNECT 'knkumar';
     EXEC SQL SELECT BOOKS.TITLE INTO :title
     FROM BOOKS,$STUDDBE.STUDENTS S,ISSUE
     WHERE BOOKS.ACCNO = ISSUE.ACCNO AND
       ISSUE.IDNO=S.ROLLNO AND S.DEPT = 'CS';
     EXEC SQL BEGIN;
printf("Book Title is %s\n",title);
     EXEC SQL END;
     EXEC SQL COMMIT;
     EXEC SQL DISCONNECT;
}
```

Answer:

AI and Speech